**URI**: scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]

**GET**: Pede uma representação do recurso indicado na URI. Não deve ter qualquer efeito sobre os dados

**HEAD**: resp semelhante ao GET, mas sem o corpo da mensagem, com a linha de status e o cabeçalho.

**POST**: Pede o envio de dados ao servidor (atualização, upload,..), incluídos no corpo da mensagem

**PUT**: Pede que um recurso, ident pela URI, seja armazenado no servid. qdo uri se refere a um rec. existente, este é substituído

**DELETE**: Solicita a eliminação do ref recurso

**TRACE**: Envia o pedido de volta ao cliente p/q possa verificar se algum servidor intermédio operou alguma modificação

**OPTIONS:** Devolv a lista dos métds HTTP suportados pelo servidor

**CONNECT:** Estabelece uma ligação/túnel ao servidor indic. na URI (usado p HTTPS na travessia de proxies s/ suporte p/ SSL)

**PATCH:** Aplica modif. parciais a um detrm. recurso; P/ subs um rec., deve usar PUT; c/ exceção do POST, tds os métds devem ser idempotents (múltiplas invocações idênticas deixam o sist. no msm estado que uma única invoc.)

**COD RESP:** 1XX: ped. receb. e processo em curso ; 2XX: ped rec e process c/suces; 3XX: redirect (ações adicionais necss); 4XX: erro cliente (erro sintaxe / sem possib de ser satisf); 5XX: erro servr (ped. válido s/ poss d ser satisf)

**Aut básica**: campo Authorization no cabeçalho do pedido: **username: password**

**Base64**: 3 bytes consecutivos da msg original (24 bits) agrupados em 4 palavras de 6 bits cada

**Cookies**: o servidor envia um cookie no campo Set-Cookie no cabeçalho da resp; O cli guarda-o num fich e transmite-o em cd ped poster no campo Cookie dos cabeçalhos dos pedidos; devem ser assinados

+ JWT (JSON Web Token), Hash-based Message Authentication Code, Secure Hash Algorithm…

**URL**: protocolo://destino[:porto]/caminho[#referência]

Web Services SOAP (Simple Object Access Protocol):

**Definição**: Protoc p troca de info estruturadas em web services; **Form de msg** XML; **Protc Subjacentes**: HTTP, SMTP, TCP; **Especificações Rigorosas**: Uso de WSDL (Web Services Description Language) p/ def serviços; **Padrões**: Mais formal, padrões específicos. **Segurança**: Integração facilitada com protocolos de segurança; **Complexidade**: + complexo que REST.

Web Services REST (Representational State Transfer):

**Definição**: Arq de comunic based em princípios REST; **Form de msg**: geralm JSON/XML; **Protc Subjac**: princip HTTP; **Princípios REST**:

Recursos ident por URIs; Operações padrão (GET, POST, PUT, DELETE); Stateless; **Simplicidade**: + leve e fácil de entend; **Padrões**: - rigoroso em q SOAP. **Segurança**: may be implm c/ HTTPS e práticas web.

(a partir daqui n esta resuindo)

N-tier

**Apresentação**: interação c/o user; Pode ser padrões cm MVC p/ organz;

**Lógica**: processamento; Coordena a comn entre a cam. apresentação e a de dados.

**Dados**: Gere o armazenamento e recup de dados. Pode incluir sist. de bd.

**Vantagens**: adapt fácil a mudanças s/ impact td o sist. manut e gest eficientes. Melhor escalabilidade e desemp distrib.

**Desafios**: necs de equil o nº de camadas p/ evitar complexidade excessiva. Considerações de desemp nas comunicações entre camadas. arquitetura é valiosa p/ desenv eficiente, especialmente em ambi empresariais.

JDBC (Java Database Connectivity): Resumo

JDBC é uma API p/ acesso a dados armazenados em bancos de dados relacionais.

**Operações Básicas**: 3 principais oper: ligação a fontes de dados, submissão de comandos SQL e devolução/processamento de respostas.

**Exemplo Elementar:** Uso de Connection, Statement, e ResultSet p/ conectar e executar consultas SQL em um banco de dados.

**Arquitetura de Acesso 2-Tier e 3-Tier:** Suporte p/ ambas as arquiteturas, onde 2-Tier envolve interação direta da aplicação com a fonte de dados e 3-Tier introduz uma camada intermédia.

**5. Drivers JDBC:**

**Tipo 1:** Mapeamento entre a JDBC e outra API de acesso a dados. Geralmente suportado por uma biblioteca nativa → portabilidade limitada. Não é habitualmente usado

**Tipo 2:** Parcialmente desenvolvidos em Java e código nativo.API JDBC →API C/C++ específico de cada SGBD.Portabilidade limitada. Usado quando não existem soluções de tipo 3 e 4 disponíveis

**Tipo 3:** Desenvolvido em Java. Permite aceder a vários SGBD em simultâneo(pref qd se usa esse caso). Comunica com um servidor intermédio (middleware) através de um protocolo de comunicação independente de qualquer SGBD. O servidor intermédio é que comunica com a fonte de dados

**Tipo 4:** Desenvolvidos em Java (pref qdo se acede a um único SGBD). A app comunica diretamente com a fonte de dados através do respetivo protocolo proprietário. Protocolos de comunicação com suporte p/ as fases de conexão, encriptação, compressão e troca de comandos e respostas.

**JDBC - Classes e interfaces:**

Driver: interface com métds p/ gerir a comunicação com o SGBD

DriverManager: classe p/ gerir um lista de drivers

Connection: interface p/ contactar o SGBD

Statement: interface p/ submeter pedidos SQL

ResultSet: interface p/ aceder às respostas aos pedidos

SQLException: classe p/ gerir erros que surjam em aplicações de BD

**executeUpdate() em JDBC:** Métd p/ executar instruções SQL de INSERT, UPDATE e DELETE, retornando o número de linhas afetadas.

**try-with-resources:** Uso recomendado p/ garantir o fechamento adequado de recursos como conexões, declarações e resultados.

O JDBC é uma ferramenta essencial p/ desenvolvedores Java que desejam interagir com bancos de dados relacionais, proporcionando uma interface eficiente e poderosa p/ consultas e manipulação de dados.

**sockets**

java.net.InetAddress

getAddress(), getHostAddress(), e getHostName()

Protocolo UDP:

não orientado a ligação.

Envio e receção de datagramas independentes.

Uso das classes DatagramPacket e DatagramSocket para representar e enviar/receber datagramas UDP.

Uma imagem com texto, captura de ecrã, Tipo de letra, file

Descrição gerada automaticamente

datagram socket

socket.func()

send(DatagramPacket packet):

receive(DatagramPacket packet):

close():

getLocalAddress():

getLocalPort():

setReceiveBufferSize(int length):

getReceiveBufferSize():

setSendBufferSize(int length):

getSendBufferSize()

setSoTimeout(int duration)

getSoTimeout

Protocolo TCP:

Orientado a ligação, comunicações ponto-a-ponto.

Uso de java.net.Socket para envio e receção de bytes.

java.net.ServerSocket para aceitação de pedidos de ligação.

Try-with-resources:

Try{socket s..} -> try(socket s…){}

Recurso preferível para garantir o encerramento automático de recursos, como sockets.

Aplicações Multicast:

IP do tipo difusão, todas as máquinas do domínio de difusão recebem os datagramas

IP do tipo multicast (classe D), apenas recebem as máquinas que se tenham registado no respetivo grupo/endereço

Utilização do protocolo UDP para comunicação multicast.

Classe MulticastSocket

joinGroup e leaveGroup

**Serialização de Objetos:**

Serialização e desserialização de objetos para comunicação em rede.

Uso de ObjectInputStream e ObjectOutputStream em sockets TCP e UDP.

Udp

bOut = new ByteArrayOutputStream();

out = new ObjectOutputStream(bOut);

out.writeObject(objectToTransmit);

//out.writeUnshared(objectToTransmit) //for avoiding caching issues

out.flush();

packet = new DatagramPacket(bOut.toByteArray(), bOut.size(), serverAddr,

serverPort);

s.send(packet);

Ttpc

s = new Socket(serverAddr, serverPort);

//TRANSMIT OBJECT

in = new ObjectInputStream(s.getInputStream());

out = new ObjectOutputStream(s.getOutputStream());

out.writeObject(objectToTransmit);

//out.writeUnshared(objectToTransmit) in order to avoid caching issues

out.flush();

//RECEIVE OBJECT

returnedObject = (MyClass)in.readObject();

Timeouts e Exceções:

Manipulação de exceções específicas, como SocketException.

Tratamento de timeouts durante operações de sockets.

**Threads**

**Grupos de Threads:**

**Objetivo**

Permitir a organização e manipulação de threads em conjuntos.

Facilitar a aplicação de operações em várias threads simultaneamente.

**Métodos Principais:**

ThreadGroup(String name) e ThreadGroup(ThreadGroup parentGroup, String name): Criam grupos de threads.

activeCount(): Retorna o número de threads ativas no grupo e seus subgrupos.

enumerate(Thread[] threadList): Copia as referências das threads ativas no grupo para um array.

destroy(): Destrói o grupo e subgrupos, desde que não haja threads ativas.

list(): Escreve informações sobre o grupo no System.out.

**Comunicação entre Threads:**

**Objetivo:**

Estabelecer mecanismos para que threads compartilhem dados e coordenem suas execuções.

**Métodos e Técnicas:**

Uso de membros públicos e métodos em objetos compartilhados.

Utilização de pipes para comunicação unidirecional.

Notificações usando Object.wait(), Object.notify(), e Object.notifyAll().

**Prioridades de Threads:**

**Objetivo:**

Gerenciar a prioridade de execução das threads em ambientes multiprocessador.

**Detalhes**:

Prioridades variam de 1 (mais baixa) a 10 (mais alta).

Thread.MAX\_PRIORITY (10), Thread.NORM\_PRIORITY (5), Thread.MIN\_PRIORITY (1).

**Servidor HTTP Concorrente**:

**Objetivo**:

Implementação de um servidor HTTP/1.0 simples, concorrente e baseado em threads.

**Funcionalidades**:

Tratamento de solicitações GET.

Manipulação de arquivos, verificação de permissões e envio de respostas HTTP.

Cada solicitação é atendida por uma thread dedicada.

**RPC (Remote Procedure Call)**

**Definição e Utilização:**

RPC é um paradigma de programação que permite a execução remota de procedimentos ou funções.

Desenvolvedores podem chamar funções em um computador remoto como se fossem funções locais.

**Elementos Principais:**

Client Stub: Age como um proxy no lado do cliente, fornecendo a interface de programação esperada.

Server Stub (Skeleton): Atua como intermediário no lado do servidor, implementando a interface remota.

**Funcionamento:**

O cliente invoca um procedimento remotamente.

O Stub do cliente envia a solicitação para o Stub do servidor através da rede.

O Stub do servidor encaminha a solicitação para a implementação real no servidor.

O resultado é enviado de volta ao cliente através dos Stubs.

**Java RMI (Remote Method Invocation):**

**Definição e Utilização:**

Java RMI é uma tecnologia que permite a invocação remota de métodos em objetos Java.

Métodos invocáveis remotamente são descritos por interfaces Java.

**Arquitetura:**

Envolvimento de servidores, clientes e RMI Registry para registro e pesquisa de objetos remotos.

Uso de Stubs (proxies) e Skeletons (servidores) para comunicação.

**Serialização:**

Dados (argumentos e resultados) são serializados antes de serem transmitidos pela rede.

Desenvolvimento RMI:

Passos incluem definição de interfaces remotas, implementação de classes concretas, produção de Stubs e Skeletons, e configuração do ambiente.

**CORBA (Common Object Request Broker Architecture):**

**Definição e Origem:**

Arquitetura para invocação remota de objetos distribuídos.

Desenvolvida pelo Object Management Group (OMG).

**Compatibilidade e Flexibilidade:**

Compatível com várias redes, sistemas operativos e linguagens de programação.

Permite a implementação de serviços e clientes em diversas linguagens.

**IDL (Interface Definition Language):**

Descreve serviços CORBA.

Suporta tipos de dados primitivos e complexos.

**Arquitetura CORBA:**

ORB (Object Request Broker): intermediário entre clientes e objetos remotos.

IIOP (Internet Inter-ORB Protocol): protocolo para redes TCP/IP.

Servants: implementam os serviços CORBA.

Serviço de Nomeação: regista e localiza servants.

**Serviços CORBA:**

Descritos por esquemas IDL.

Realizados por servants, registados num serviço de nomeação para independência de localização.

**IDL para Java:**

Mapeamento de IDL para Java usando o IDL-to-Java compiler (idlj).

**Execução:**

Lançar serviço de nomeação (orbd.exe ou tnameserv.exe).

Lançar aplicação servidora (java LightBulbServant).

Correr clientes (java LightBulbClient).

**Resultados:**

Demonstração de invocações remotas e tratamento de exceções em objetos distribuídos.

**Conclusão:**

CORBA oferece uma abordagem flexível e interoperável para sistemas distribuídos, facilitando a comunicação entre objetos em diferentes linguagens e plataformas.

s = new Socket(serverAddr, serverPort); //TRANSMIT OBJECT

in = new ObjectInputStream(s.getInputStream());

out = new ObjectOutputStream(s.getOutputStream());

out.writeObject(objectToTransmit); //out.writeUnshared(objectToTransmit)

out.flush(); //RECEIVE OBJECT

returnedObject = (MyClass)in.readObject();